# *A simple method for rejection sampling efficiency improvement on SIMT architectures*



**Citation:** Statistics and Computing. 2021 Mar 30;31(3):30

**As Published:** https://doi.org/10.1007/s11222-021-10003-z

**Publisher:** Springer US

**Persistent URL:** https://hdl.handle.net/1721.1/132084

**Version:** Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

# A simple method for rejection sampling efficiency improvement on SIMT architectures

**Cite this article as:** Gavin Ridley and Benoit Forget, A simple method for rejection sampling efficiency improvement on SIMT architectures, Statistics and Computing https://doi.org/10.1007/s11222-021-10003-z

# A Simple Method for Rejection Sampling Efficiency Improvement on SIMT Architectures

Gavin Ridley *[1] and Benoit Forget †[1]

[1]Department of Nuclear Science & Engineering, Massachusetts Institute of Technology

February 16, 2021



## 1 Abstract

We derive a probability distribution for the possible number of iterations required for a SIMT (single instruction multiple thread) program using rejection sampling to finish creating a sample across all threads. This distribution is found to match a recently proposed distribution from [6] that was shown as a good approximation of certain datasets. This work demonstrates an exact application of this distribution. The distribution can be used to evaluate the relative merit of some sampling methods on the GPU without resort to numerical tests. The distribution reduces to the expected geometric distribution in the single thread per warp limit. A simplified formula to approximate the expected number of iterations required to obtain rejection iteration samples is provided. With this new result, a simple, efficient layout for assigning sampling tasks to threads on a GPU is found as a function of the rejection probability without recourse to more complicated rejection sampling methods.

## 2 Introduction

Due to advances in both FLOPs per unit power and cost over CPUs, GPUs are displacing CPU computations at an increasing pace. These advances have manifested in the construction of multiple GPU-based HPC machines around the world: Satori, Summit, etc. While these machines may excel in tasks like deep learning and computational fluid dynamics, leveraging the full computational potential of GPUs can be difficult for certain algorithms. To understand the cause of this issue on GPUs, we first provide some background on computational parallelism.

Flynn's taxonomy of computer architectures [9] broadly classifies the commonly used shared memory parallel computer architectures as either single instruction, multiple data or multiple instruction, multiple data. These are respectively abbreviated as SIMD and MIMD, and both types of parallelism are executed on comtemporary desktop computers. Multicore computer processors realize the MIMD approach in that each core can execute different instructions on multiple data simultaneously. Typically, each of these cores can realize parallelism individually by the use of special SIMD instructions which carry out the same operation on several adjacent pieces of data. These operations are typically restricted to addition, multiplication, or rudimentary logic operations. By using these special instructions on modern processors, throughput of numerical programs can be increased by a factor approaching the SIMD data length. For more information on parallel computing in the context of statistics, the article [2] expounds the aforementioned concepts further.

GPUs operate using a so-called SIMT, or single instruction multiple thread, model [14] which means that many threads can carry out a single instruction at a time on arbitrary data. This is distinctly more flexible

*ridley@mit.edu
†bforget@mit.edu

than SIMD (single instruction multiple data), a commonly available form of parallelism on modern CPUs. In SIMT, a full-featured instruction set may be executed on data in parallel; this contrasts from CPUs where specific instructions must be used to leverage SIMD. Because of this, branching statements can be used in parallel with ease, and although each branch must execute in lockstep thus degrading performance, some degree of parallelism is achieved. For instance, thirty-two threads may encounter two branches where 15 will take one and 17 the other. First one branch could execute on 15 threads, then the next on 17 threads. This allows a degree of programming versatility superseding SIMD [1].

The group of threads each executing an instruction is called a warp, and there are typically thirty-two threads per warp on an Nvidia GPU [7]. As previously mentioned, it is commonplace for a warp of threads to encounter a point in the program where some threads enter an if-block, and others do not. This situation is called *warp divergence*, and should be avoided if possible since it degrades the computational throughput of the GPU. Warp divergence may be impossible to avoid in rejection sampling algorithms where some of the threads in a warp may have to sample again in order to obtain an accepted sample. This paper finds a closed form result for how the number of iterations required to complete rejection sampling is distributed on a warp as a function of the number of threads per warp and the rejection probability. This distribution is found to exactly match the exponentiated geometric distribution proposed by [6], which to the authors' knowledge has not been previously found to exactly govern any other processes.

Although rejection sampling typically takes more time to draw a sample on a computer than other methods, the method is robust and can sample any distribution for which the maximum probability is known. To restate the definition of rejection sampling given by [8], rejection sampling from a distribution $X$ with support in $\mathbb{R}^n$ with density $f$. Then, if $g$ is another distribution in the same space we can draw samples from $f$ by the following procedure if

$$\exists c \geq 1, c \in \mathbb{R} \quad \text{s.t.} \quad f(x) \leq cg(x) \forall x \in \mathbb{R}^n. \tag{1}$$

The rejection sampling procedure, if this condition is met and $c$ accordingly found can then proceed according to Algorithm 1 to find one sample from the distribution $X$. Algorithm 2 shows how this can be modified to execute in parallel.

**repeat**
  sample $U$ uniformly from $[0, 1]$;
  sample $X$ with density $g$;
**until** $Ucg(X) \leq f(X)$;
**return** $X$;

**Algorithm 1:** Rejection sampling requires a loop which terminates stochastically.

**repeat**
  **if** $U_i cg(X_i) \leq f(X_i)$ **then**
    continue;
  **end**
  **else**
    sample $U_i$ uniformly from $[0, 1]$;
    sample $X_i$ with density $g$;
  **end**
**until** $U_i cg(X_i) \leq f(X_i) \forall i \in [1, t]$;
**return** $X_i$;

**Algorithm 2:** SIMT-parallel rejection sampling requires a loop which terminates stochastically on each of the $t$ threads of a warp.

Understanding the performance of rejection sampling on SIMT architectures can be impactful because rejection sampling is found in myriad applications. Rejection methods are most commonly employed in Monte

[1] Masked SIMD instructions can also achieve this, but the programming is more cumbersome and these instructions are not always available.

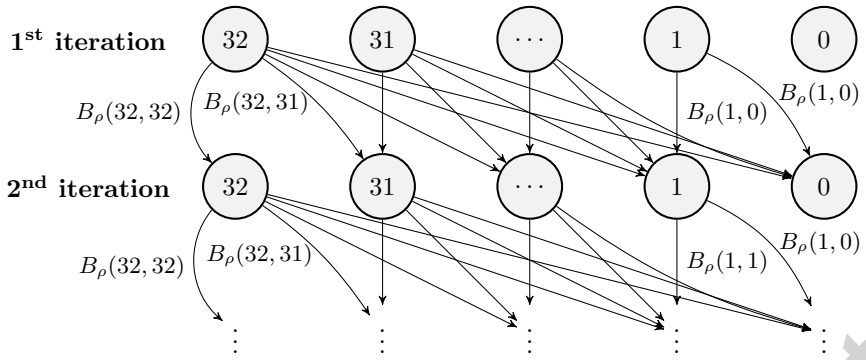



Figure 1: Infinite Markov chain representation of transitions between possible amounts of threads awaiting an accepted sample. Two iterations of a thirty-two thread rejection sampling algorithm are depicted. Edges are the probability of transitioning between each state, given by the binomial distribution $B_\rho(i,j)$. The relevant value of the binomial distribution is placed left of a few edges above.

Carlo methods. Rejection is used nearly ubiquitously to sample from the gamma distribution, following the method prescribed by [12]. The sampling of distributions can be a performance bottleneck for certain problems like deep belief networks where GPUs are used to generate millions of samples as quickly as possible. In addition, rejection sampling may be used in Monte Carlo particle transport simulations, e.g., OpenMC [19] uses rejection to sample outgoing scattering angle distributions for particle simulations where particles take on discrete energies and for sampling the resonance upscattering effect [20] encountered in continuous particle energy simulations. Several recent developments like [21] and [3] extensively employ rejection sampling in their proposed algorithm. In light of this, a theoretical model for the expected decrease in speed of SIMT versus MIMD evaluation of a rejection sampling method would be useful.

## 3 Derivation of the distribution

Let the rejection probability for a sample be $\rho$. Then, among $t$ threads, the probability of $k$ threads being rejected to require another sample is given by the binomial distribution:

$$B_\rho(t,k) = \binom{t}{k}\rho^k(1-\rho)^{t-k} \tag{2}$$

Using this, we can use induction to predict how the number of threads awaiting an accepted sample changes with each iteration. $t$ is the number of threads initially seeking a sample via rejection, and call $t_1, t_2, \cdots$ the number of threads yet to obtain a sample at each of the successive sampling iterations.

Now, consider the random variables $t_n$ and $t_{n+1}$. We can see from Fig. 1 that:

$$t_{n+1}|t_n \sim B_\rho(t_n, t_{n+1}) \tag{3}$$

Moreover, if $t_n$ has the distribution

$$t_n \sim B_{\rho^n}(x, t_n) \tag{4}$$

it is then true that

$$t_{n+1} \sim B_{\rho^{n+1}}(x, t_{n+1}) \tag{5}$$

From here, it is straightforward to establish the distribution of each $t_n$ inductively. Using the base case:

$$t_1 \sim B_\rho(t, t_1) \tag{6}$$

We then immediately see from the previously stated inductive rule:

$$t_n \sim B_{\rho^n}(t, t_n) \tag{7}$$

We can now calculate the probability of termination of the sampling routine in $n$ steps. To be precise, the quantity in question is $P[t_n = 0|t_{n-1} > 0]$. Computing this by summing all of the independent probabilities in question gives:

$$\begin{aligned} P[t_n = 0|t_{n-1} > 0] = p_{\rho,t}(n) &= \sum_{i=1}^{t} (1-\rho)^i B_{\rho^{n-1}}(t,i) \\ &= \sum_{i=0}^{t} \binom{t}{i} \left(\rho^{n-1} - \rho^n\right)^i \left(1-\rho^{n-1}\right)^{t-1} - \left(1-\rho^{n-1}\right)^t \end{aligned} \tag{8}$$

After applying the binomial theorem once more, this results in the desired distribution for the probability of $t$ SIMT threads taking $n$ iterations to sample some distribution using a rejection method with rejection probability $\rho$:

$$\boxed{p_{\rho,t}(n) = (1-\rho^n)^t - (1-\rho^{n-1})^t} \tag{9}$$

This discrete distribution has been plotted for a few values of the parameters in Fig. 2. Low $\rho$ values are plotted because these are commonly encountered in ziggurat-type algorithms [13]. This distribution has a cumulative distribution function (CDF) of $(1-\rho^n)^t$. This distribution is identical to the one recently proposed by [6], named the exponentiated geometric distribution. This distribution was originally proposed as a modification to the geometric distribution, found simply by exponentiating the CDF of the geometric distribution. To the extent of the authors' knowledge, the exponentiated geometric distribution has thus far only been proposed to fit certain data better than other distributions. This may be the first known case where the exponentiated geometric distribution exactly governs the underlying statistical process.

We have verified in numerical experiment that this distribution indeed governs the rejection sampling process on the GPU with a simple CUDA program. A surrogate rejection sampling process has been used, where a rejection loop exits with probability $1-\rho$. The code for this numerical experiment can be found in Appendix A, and its results compared to the new theoretical result are illustrated by Fig. 3.

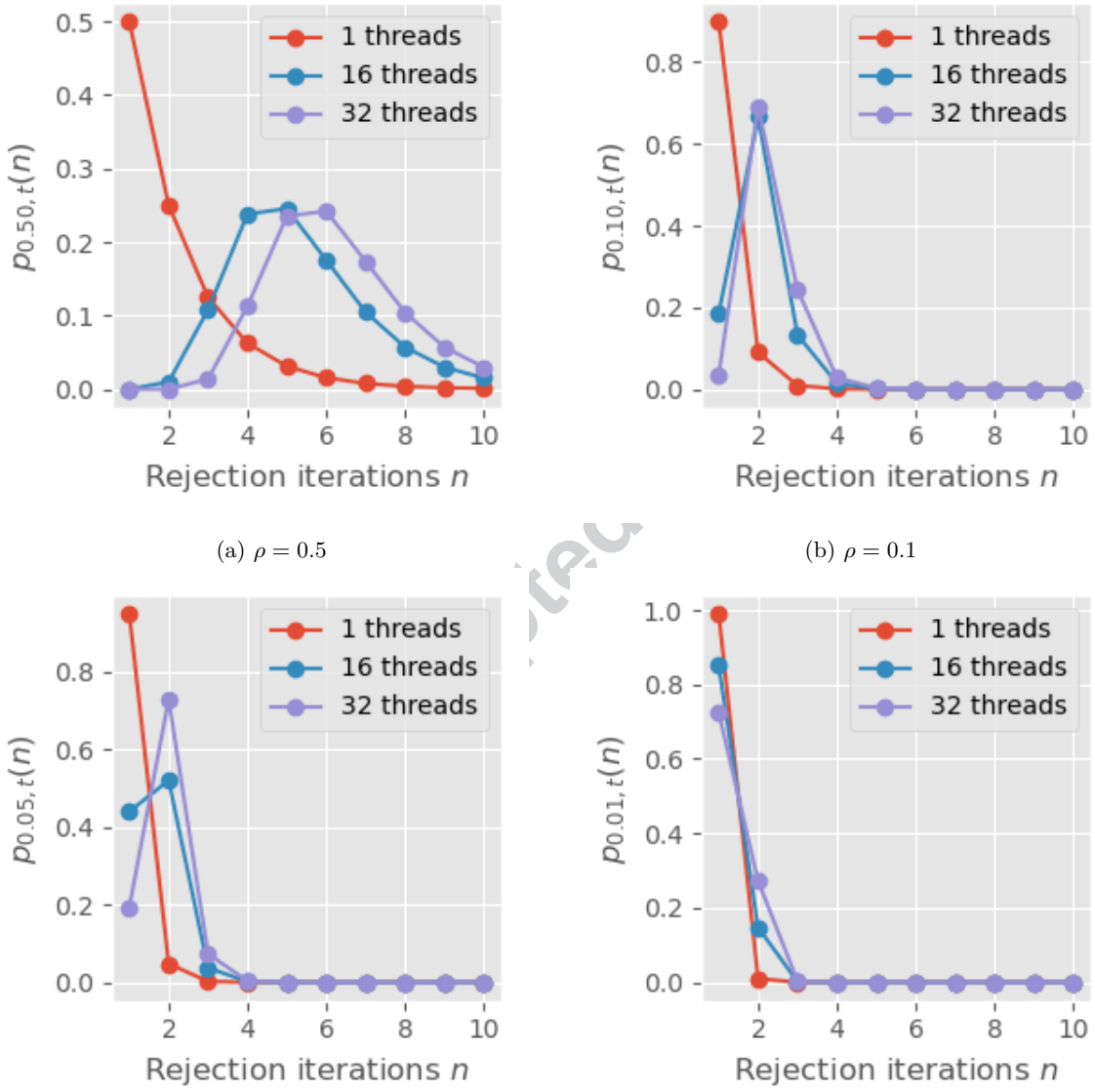
# 4 Approximate Formula for Mean of The Distribution

The work of [6] provides a convenient numerical expression for all of the moments of the exponentiated distribution. The formula provided contains a summation over infinitely many indices which does not clearly evince any information about the sensitivity of the mean with respect to the distribution parameters. Moreover, we also point out that while finding the mean of this distribution is indeed tractable in closed form, we have verified using computer algebra software for the case of interest $t = 32$ that this takes the form of a three-hundred and twenty-third order rational polynomial function in $\rho$. Obviously, not much is afforded here in terms of intuitive understanding of the behavior of the mean with respect to $\rho$ and $t$.

As a result of this, we have derived a simple expression for the mean which can guide intuition about the behavior of the exponentiated geometric distribution upon changing its parameters. This is done in two parts: firstly for small values of $\rho$ as are reminiscent of ziggurat-type algorithms, and secondly for larger values of $\rho$. This is because the exponentiated geometric transitions from being a monotonically decreasing distribution to a unimodal distribution for $\rho > 2^{1/t} - 1$, as shown by [6]. Two separate formulas can approximate these regions with excellent accuracy.

## 4.1 Approximate Mean in Unimodal Region

It was proved by [5] that the mean of a unimodal distribution always lies within 1.74 standard deviations of the mode. Regarding the distribution discussed here, [6] proved unimodal behavior for $\rho > 2^{1/t} - 1$, so we know the mean could be well-approximated by the mode for $\rho > 2.2\%$ in the 32 thread per warp case which is ubiquitously found in GPUs. Finding a mode here is much easier than finding the mean, which cannot be described in terms of elementary functions for general $t$, as far as we have found. Despite this inconvenience, Fig. 5 shows that the mode approximates the mean quite acceptably. Approximating the mean more accurately has been done, but we found that this results in complicated combinations of special




(a) $\rho = 0.5$

(b) $\rho = 0.1$

(c) $\rho = 0.05$

(d) $\rho = 0.01$

Figure 2: Probability distributions of completion of the rejection sampling algorithm in a given number of iterations for $\rho = 0.5$ (top left), $\rho = 0.1$ (top right), $\rho = 0.05$ (bottom left), and $\rho = 0.01$ (bottom right).

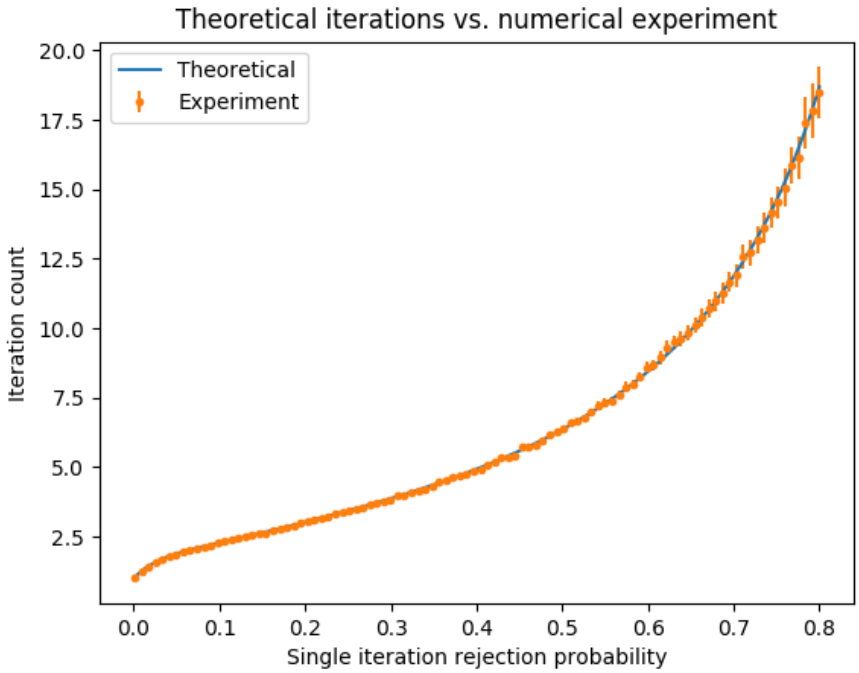



Figure 3: The code of Appendix A was used to verify the predicted mean iteration count by Eq. 9, with which excellent agreement has been obtained.



functions like the polylog and hypergeometric functions which lend little insight to the problem at hand, and thus not discussed.

To begin, note that the binomial expression, with $t$ large, is well-approximated by:

$$(1-\rho^n)^t \approx \exp(-\rho^n t) \tag{10}$$

This approximation comes from observing that because $e^x = \lim_{N\to\infty}(1+\frac{x}{N})^N$, if we define $x' = -\rho^n t$, the expression above is simply $(1+\frac{x'}{t})^t$. From this definition of the exponential, we can see that this may arbitrarily closely approximate $e^{x'}$ as $t$ grows.

Using this, the exponentiated geometric distribution is well-approximated by the following continuous distribution, so long that $t$ is large:

$$p_{\rho,t}(n) \approx \left(\exp(-\rho^n t) - \exp(-\rho^{n-1} t)\right) \tag{11}$$

Then, note that this continuous approximation very nearly preserves normalization:

$$\begin{aligned}\int_0^\infty \left(\exp(-\rho^n t) - \exp(-\rho^{n-1} t)\right) \mathrm{d}n &= \frac{1}{\ln(\rho)} \int_1^0 \left(\exp(-ut) - \exp(-ut/\rho)\right)/u \,\mathrm{d}u \\ &= 1 - \frac{1}{\ln(\rho)}\left(\mathrm{Ei}(-t) - \mathrm{Ei}(-t/\rho)\right)\end{aligned} \tag{12}$$

Where the substitution $u = \rho^x$ was used, and $\mathrm{Ei}(\cdot)$ is the exponential integral function, defined in almost any table of integrals, e.g. [1]. Fig. 4 clearly demonstrates the adequacy of this approximation for $t > 16$ in terms of its preservation of normalization by plotting the difference between 12 and unity.

With this in mind, this value of $n$ which maximizes Eq. 11 is expected to approximate the mean of the discrete distribution in question:

$$\mu \approx \frac{\ln\left(\frac{2(\rho-1)t}{\ln\rho}\right)}{\ln\left(1/\rho\right)} + 1 \tag{13}$$

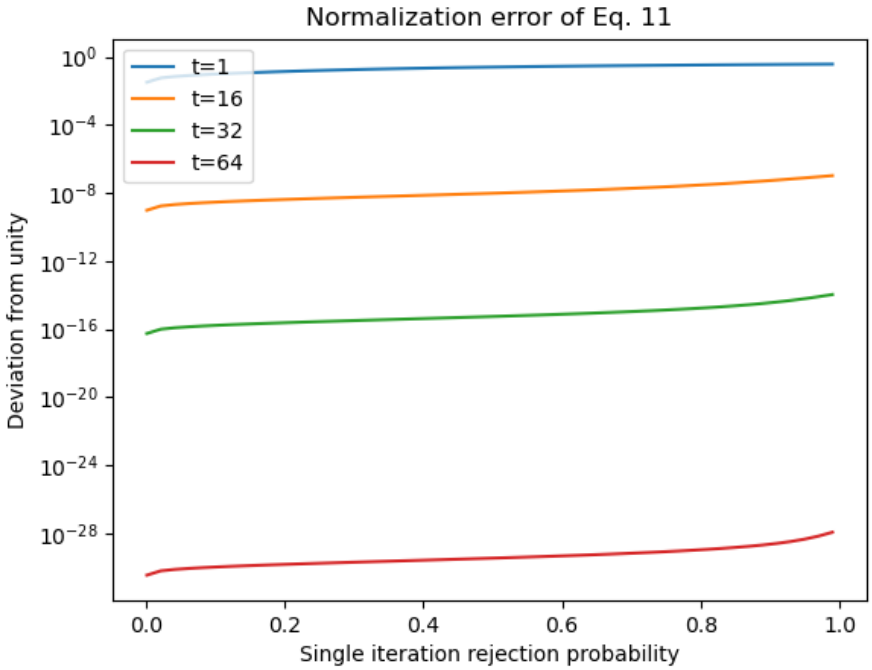



Figure 4: This depicts the difference between the integral of Eq. 11 over its domain and unity. The preservation of normalization by using the approximation Eq. 11 can be seen to be excellent for $t > 16$.

From this, we can see that the expected number of iterations required to exit the parallel rejection sampling loop grows logarithmically with the number of threads for $r$ sufficiently large.

## 4.2 Approximate Mean in Monotically Decreasing Region

In the monotonically decreasing region, i.e., where $\rho < 2^{1/t} - 1$, the derivation of an approximate formula for the mean is much simpler and can be done based on the fact that $\rho$ will be quite small here. The derivation proceeds by considering the exact expression for the mean in terms of an infinite sum of integers times their corresponding probabilities, regrouping some terms in that series, and approximating $\rho^k \approx 0$ for higher order terms in the sum, where $k$ is a small integer. So, the exact mean is:

$$\begin{aligned}\mu = \sum_{i=1}^{\infty} i\left((1-\rho^i)^t - (1-\rho^{i-1})^t\right) = (1-\rho) + \\ 2\left((1-\rho^2) - (1-\rho)\right) + \\ 3\left((1-\rho^3) - (1-\rho^2)\right) + \\ 4\left((1-\rho^4) - (1-\rho^3)\right) + \\ \ldots\end{aligned} \tag{14}$$

Note how in the second line, we see $-2(1-\rho)$, which cancels out with the term in the first line. This cancellation can be carried out $N$ times in order to see that the mean is:

$$\mu = \lim_{N\to\infty}\left(N(1-\rho^N)^t - \sum_{i=1}^{N}(1-\rho^i)^t\right) \tag{15}$$

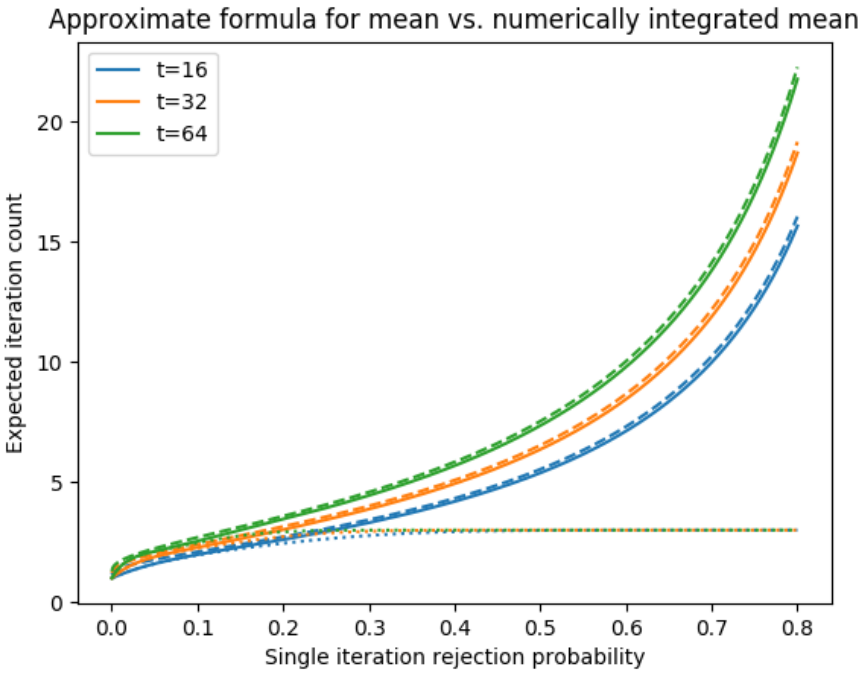



Figure 5: The dashed curve is from the large $\rho$ Eq. 13, dotted curves are the small $\rho$ Eq. 16, and solid curves are the true mean for a few $t$ values.

The key to obtaining a good approximation is to take $k$ terms of the first sum, and suppose that the remaining terms are close to one since $\rho^k$ is very small. This allows a cancellation with the increasing term on the right, and for $k = 2$ gives:

$$\mu \approx 3 - (1-\rho)^t - (1-\rho^2)^t \tag{16}$$

With that, an accurate piecewise approximation to the mean of this distribution is:

$$\mu \approx \begin{cases} 3 - (1-\rho)^t - (1-\rho^2)^t & \rho < 2^{1/t} - 1 \\ \frac{\ln\left(\frac{2(\rho-1)t}{\ln\rho}\right)}{\ln(1/\rho)} + 1 & \text{else} \end{cases} \tag{17}$$

Both formulas have been plotted in Fig. 5, with the approximate mean formula compared to the actual numerically calculated mean for a few different values of the thread count. Fig. 6 shows these formulas for small values of $\rho$, where the second formula can be seen to give great accuracy.

## 5 A More Efficient Sampling Strategy

Observing from Fig. 5 that it is expected to take over seven iterations on average for thirty-two threads to obtain a sample with a 50% rejection probability, which compares unfavorably with the expected two iterations encountered in MIMD rejection sampling algorithms, it is natural to suppose that a more efficient algorithm could perhaps be obtained if, say, only 16 samples are produced by the warp in groups of two threads working to obtain a sample. As a further example, consider an extreme case like a 99% rejection probability. It would be expected that dedicating thirty-two threads to each finding a single accepted sample would be more efficient than waiting on thirty-two threads to each obtain their own sample, which, according to the formulas above, would take around 400 iterations. In this case, the probability of all threads rejecting their sample would simply be $\rho^{32}$, in which case a single sample would be obtained in $1/(1-\rho^{32}) \approx 3.6$ iterations. As such, the samples obtained per rejection sampling iterations grows from 0.08 to 0.28.

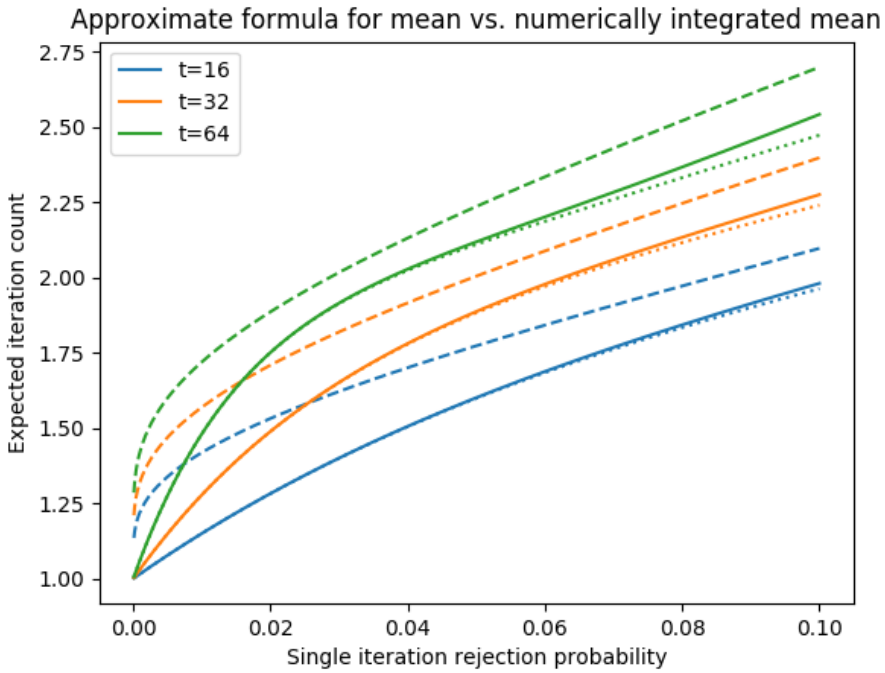



Figure 6: This is Fig. 5 but with the abscissa zoomed to the $\rho \in [0, 0.1]$ interval to show how the dotted curve Eq. 16 is best in this range.



Algorithm 3 concisely describes our proposed modified rejection sampling method to be carried out on each thread group of a SIMT device. This section explores when strategies like this are more efficient than the standard algorithm.

Most rejection sampling algorithms are designed such that the proposal distribution yields a low rejection probability, which is most computationally efficient. As such, it remains in question whether the investigation of high rejection probabilities are worthwhile, considering that algorithms such as the ziggurat method [13] typically yields rejection probabilities less than five percent with a properly designed proposal distribution table. Similarly, the commonly employed rejection sampling scheme for generating gamma variates by [12] has a rejection probability less than five percent.

Some more specialized algorithms, in fact, must cope with higher rejection probabilities. From nuclear engineering, the Doppler broadening rejection correction algorithm [20], which can form a computationally significant portion of a Monte Carlo neutron transport simulation, exhibits rejection probabilities over 99% for some parts of the simulation as shown in [18]. Similarly, [11] presents a problem which exhibits rejection rates over 99.99%, along with a solution to that inefficiency. It is also commonly noted that high dimensional probability distributions typically exhibit high rejection rates, e.g. in [16].

On the ubiquitous thirty-two-thread warp, it would make sense to consider firstly thirty-two threads each attempting to obtain their own sample on each iteration. The next possible configuration would be sixteen groups of two threads where only one sample is asked of each thread pair per iteration. Following that would be eight groups of four threads, sixteen groups of two threads and finally a single group of thirty-two threads looking for a single sample. The formulas above can be used still, but $t$ is divided by the number of groups per warp and $\rho$ is raised to the power of the number of threads per group, representing the probability that each sample from the group was rejected.

The total samples obtained per rejection sampling iteration for each of these configurations have been plotted in Fig. 7, where it can be seen that small but perhaps worthwhile performance gains can be made by switching to these alternative sampling setups. Similarly, Fig. 8 plots the ratio between the optimal method out of this class of methods compared to the case of attempting to obtain a sample on each thread in the

```
repeat
    if U_i cg(X_i) ≤ f(X_i) then
        continue;
    end
    else
        foreach thread t in group n do
            sample V_t uniformly from [0, 1];
            sample Y_t with density g;
        end
        if Any T_t cg(Y_t) ≤ f(Y_t) then
            save Y_t as this group's sample
        end
    end
until all thread groups have a sample;
return X_i;
```

**Algorithm 3:** Our proposed modified rejection sampling algorithm which can produce more samples per rejection sampling iteration. There are $n$ thread groups which each produce a sample.

Table 1: The optimal number of threads per sample to use are given here, in addition to the average expected speedup compared to a single thread per sample for this region, corresponding to divisions by vertical lines in Fig 7.

| Threads per sample | Maximum efficient $\rho$ (%) | Average Speedup |
|---|---|---|
| 1 | 12.88 | 1 |
| 2 | 42.71 | 1.01 |
| 4 | 71.70 | 1.14 |
| 8 | 88.37 | 1.39 |
| 16 | 95.76 | 1.84 |
| 32 | 1 | 2.65 |

warp. Table 1 gives the specific rejection probabilities where it is optimal to switch to a different layout of thread grouping.

# 6 Discussion

Results have been obtained which predict the performance of rejection sampling in the context of SIMT parallel architectures. While these results do not apply to MIMD parallelism that does not suffer from the necessity of carrying out the same instructions in lockstep, these results do translate to SIMD parallelism where masked instructions are supported. In that case, the number of samples to obtain would correspond to the number of floating point numbers operated on by the SIMD instructions. In the case of the latest Intel AVX-512 instructions [10], which can operate on sixteen single precision numbers at once, this would correspond to the $t = 16$ case. Masked SIMD instructions, which allow SIMD instructions on a subset of the data, could be used to control the exit from the rejection sampling loop. This has not been explored by the authors and could be examined in future work.

The proposed algorithm of this paper is certainly sub-optimal in terms of obtaining the maximum sampling rate for the sampling of one distribution repeatedly on GPU. If this were the desired objective, persistent threading strategies for this task are surely superior. Such methods are detailed by [4] and [15], and have been conclusively shown to improve performance in programs suffering from warp divergence.

The method we present is strong in two distinct ways, however. Firstly, the implementation of this method is very simple, and performance improvements can be made to a GPU Monte Carlo program with little programming effort. Only a few lines of warp-level voting logic and warp-shared memory need be added to implement the method. We expect that statistical software relying on a computationally intensive

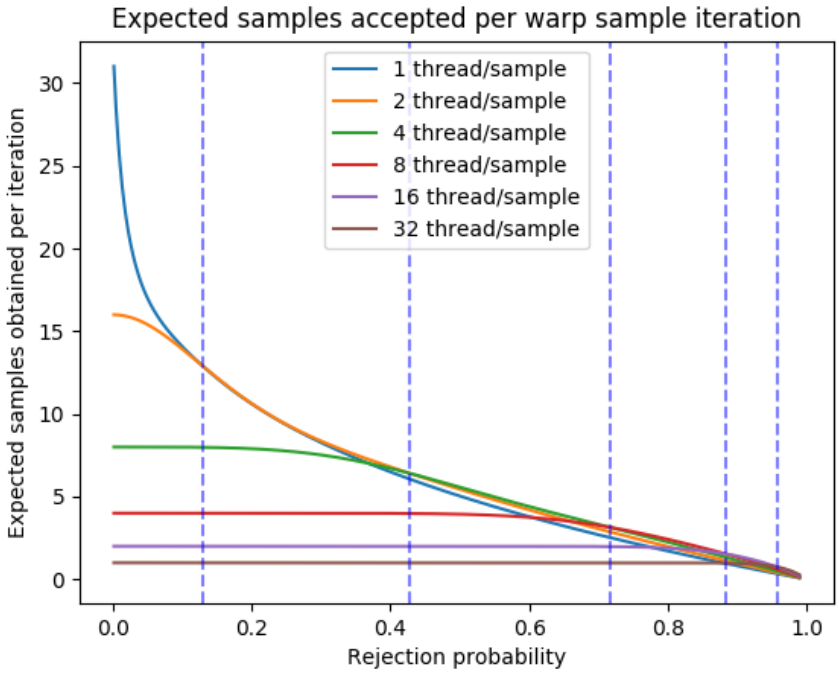



Figure 7: Different thread grouping patterns can obtain better sampling performance per unit of GPU work depending on the rejection probability. The dashed vertical lines indicate a location where solid curves intersect, and switching to another grouping is desirable.

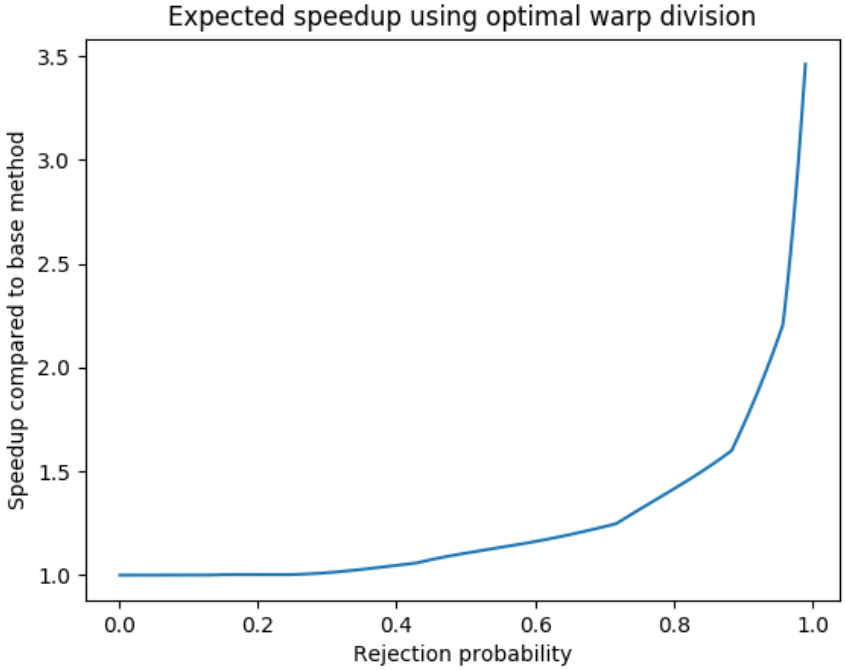



Figure 8: As a function of the rejection probability, this plots the ratio between the sampling rate of the best thread grouping of Fig. 7 and the base case where each thread is responsible for finding a sample.

rejection loop could quickly add this grouped rejection sampling algorithm and easily obtain performance improvements.

Secondly, the proposed method method has been designed to fit into larger Monte Carlo programs which only use rejection sampling as an intermediate but perhaps computationally intensive step. Because different warps may be attending to other parts of the sampling routine of interest, individual warps may be responsible for carrying out a rejection sampling part of the calculation. Because of this, the lauded persistent threading strategies described by works like [4] and [15] were not considered because that very warp may be expected to perform a completely different task in the next stage of the Monte Carlo computation. These are the conditions encountered in Monte Carlo neutron transport simulations that the authors had in mind while developing this method, and we found that similar conditions prevail in large statistical calculations like that presented in [15].



# 7 Conclusion

We have shown that the exponentiated geometric distribution [6] exactly governs the statistics of rejection sampling performed by a warp of threads on GPU, where each thread within the warp must perform the same instruction, known as SIMT. The exponentiated geometric distribution has only been shown to fit certain data sets better than other distributions thus far. The findings of this paper suggest that similar phenomena may be identifiable as being distributed according to the exponential geometric.

An approximate formula for the expected number of iterations taken by a thread warp to complete a rejection sampling iteration has been provided, and this formula has been numerically verified to accurately model the expected iteration count for rejection probabilities greater than about 2%. For smaller rejection probabilities, the formula does not work as well, and an alternative formula has been provided.

Lastly, with this theoretical result, we have provided insight on GPU rejection sampling algorithms without resort to numerical experiments. Our result has been verified with a numerical experiment, with

which excellent agreement was obtained. This new theoretical result was used to define a more efficient rejection sampling layout on GPU that is particularly effective for high rejection probability. Further work could expand on these results, particularly regarding the theoretical performance of rejection sampling algorithms that utilize sample caching as in [17].

## Acknowledgement

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

## References

[1] Milton Abramowitz and Irene A. Stegun, eds. *Handbook of Mathematical Functions: With Formulas, Graphs, and Mathematical Tables*. 0009-Revised edition. New York, NY: Dover Publications, June 1, 1965. 1046 pp. ISBN: 978-0-486-61272-0.

[2] N. M. Adams et al. "A Review of Parallel Processing for Statistical Computation". In: *Statistics and Computing* 6.1 (Mar. 1, 1996), pp. 37–49. ISSN: 1573-1375. DOI: 10.1007/BF00161572. URL: https://doi.org/10.1007/BF00161572 (visited on 09/09/2020).

[3] Amir Ahmadi-Javid and Asghar Moeini. "An Economical Acceptance–Rejection Algorithm for Uniform Random Variate Generation over Constrained Simplexes". In: *Statistics and Computing* 26.3 (May 1, 2016), pp. 703–713. ISSN: 1573-1375. DOI: 10.1007/s11222-015-9553-x. URL: https://doi.org/10.1007/s11222-015-9553-x (visited on 03/10/2020).

[4] Timo Aila and Samuli Laine. "Understanding the Efficiency of Ray Traversal on GPUs". In: *Proc. High Performance Graphics*. High Performance Graphics. Aug. 1, 2009. URL: https://research.nvidia.com/publication/understanding-efficiency-ray-traversal-gpus (visited on 09/09/2020).

[5] S. Basu and A. DasGupta. "The Mean, Median, and Mode of Unimodal Distributions:A Characterization". In: *Theory of Probability & Its Applications* 41.2 (Jan. 1, 1997), pp. 210–223. ISSN: 0040-585X. DOI: 10.1137/S0040585X97975447. URL: https://epubs.siam.org/doi/10.1137/S0040585X97975447 (visited on 02/22/2020).

[6] Subrata Chakraborty and Rameshwar D. Gupta. "Exponentiated Geometric Distribution: Another Generalization of Geometric Distribution". In: *Communications in Statistics - Theory and Methods* 44.6 (Mar. 19, 2015), pp. 1143–1157. ISSN: 0361-0926. DOI: 10.1080/03610926.2012.763090. URL: https://doi.org/10.1080/03610926.2012.763090 (visited on 02/22/2020).

[7] *CUDA Toolkit Documentation V10.2.89*. URL: https://docs.nvidia.com/cuda/ (visited on 02/13/2020).

[8] Luc Devroye. "Chapter 4 Nonuniform Random Variate Generation". In: *Handbooks in Operations Research and Management Science*. Ed. by Shane G. Henderson and Barry L. Nelson. Vol. 13. Simulation. Elsevier, Jan. 1, 2006, pp. 83–121. DOI: 10.1016/S0927-0507(06)13004-2. URL: http://www.sciencedirect.com/science/article/pii/S0927050706130042 (visited on 02/13/2020).

[9] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (Sept. 1972), pp. 948–960. ISSN: 1557-9956. DOI: 10.1109/TC.1972.5009071.

[10] *Intel® Advanced Vector Extensions 512 Overview*. URL: intel.com/avx512 (visited on 09/09/2020).

[11] Tobias Kunz, Andrea Thomaz, and Henrik Christensen. "Hierarchical Rejection Sampling for Informed Kinodynamic Planning in High-Dimensional Spaces". In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016 IEEE International Conference on Robotics and Automation (ICRA). May 2016, pp. 89–96. DOI: 10.1109/ICRA.2016.7487120.

[12] George Marsaglia and Wai Wan Tsang. "A Simple Method for Generating Gamma Variables". In: *ACM Transactions on Mathematical Software (TOMS)* 26.3 (Sept. 1, 2000), pp. 363–372. ISSN: 0098-3500. DOI: 10.1145/358407.358414. URL: https://doi.org/10.1145/358407.358414 (visited on 02/13/2020).

[13] George Marsaglia and Wai Wan Tsang. "The Ziggurat Method for Generating Random Variables". In: *Journal of Statistical Software* 5.1 (1 Oct. 2, 2000), pp. 1–7. ISSN: 1548-7660. DOI: 10.18637/jss.v005.i08. URL: https://www.jstatsoft.org/index.php/jss/article/view/v005i08 (visited on 04/11/2020).

[14] Norm Matloff. *Programming on Parallel Machines*.

[15] Lawrence Murray. "GPU Acceleration of Runge-Kutta Integrators". In: *IEEE Transactions on Parallel and Distributed Systems* 23.1 (Jan. 2012), pp. 94–101. ISSN: 1558-2183. DOI: 10.1109/TPDS.2011.61.

[16] Roger D. Peng. *6.3 Rejection Sampling — Advanced Statistical Computing*. URL: https://github.com/rdpeng/advstatcomp (visited on 09/16/2020).

[17] Avi Pfeffer. "Sampling with Memoization". In: *AAAI*. 2007.

[18] Paul Romano and John Walsh. "An Improved Target Velocity Sampling Algorithm for Free Gas Elastic Scattering". In: *Annals of Nuclear Energy* 114 (Apr. 2018), pp. 318–324.

[19] Paul K. Romano et al. "OpenMC: A State-of-the-Art Monte Carlo Code for Research and Development". In: *Annals of Nuclear Energy*. Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2013, SNA + MC 2013. Pluri- and Trans-Disciplinarity, Towards New Modeling and Numerical Simulation Paradigms 82 (Aug. 1, 2015), pp. 90–97. ISSN: 0306-4549. DOI: 10.1016/j.anucene.2014.07.048. URL: http://www.sciencedirect.com/science/article/pii/S030645491400379X (visited on 03/10/2020).

[20] W. Rothenstein. "Neutron Scattering Kernels in Pronounced Resonances for Stochastic Doppler Effect Calculations". In: *Annals of Nuclear Energy*. A Special Issue in Honour of M. M. R. Williams 23.4 (Mar. 1, 1996), pp. 441–458. ISSN: 0306-4549. DOI: 10.1016/0306-4549(95)00109-3. URL: http://www.sciencedirect.com/science/article/pii/0306454995001093 (visited on 03/10/2020).

[21] Alexander Terenin, Shawfeng Dong, and David Draper. "GPU-Accelerated Gibbs Sampling: A Case Study of the Horseshoe Probit Model". In: *Statistics and Computing* 29.2 (Mar. 1, 2019), pp. 301–310. ISSN: 1573-1375. DOI: 10.1007/s11222-018-9809-3. URL: https://doi.org/10.1007/s11222-018-9809-3 (visited on 02/13/2020).

# A Numerical Experiment Code

```
// This calculates the distribution of the number of iterations required for a warp
// of threads to complete a rejection sampling iteration.
#define WARPSIZE 32
#include <iostream>
#include <vector>

constexpr uint64_t master_seed {1};
constexpr uint64_t prn_mult   {2806196910506780709LL};   // multiplication
constexpr uint64_t prn_add    {1};                        // additive factor, c
constexpr uint64_t prn_mod    {0x8000000000000000};       // 2^63
constexpr uint64_t prn_mask   {0x7fffffffffffffff};       // 2^63 - 1
constexpr double   prn_norm   {1.0 / prn_mod};            // 2^-63

// A linear congruential random number generator
__device__ __inline__ double prn(uint64_t* seed)
{
  *seed = (prn_mult * (*seed) + prn_add) & prn_mask;
  return (*seed) * prn_norm;
}

// Each block takes on its own rejection probability
```

```
template<unsigned Warpsize>
__global__ void calculate_mean_iterations_required(double* rejection_probabilities,
                                                   double* mean_iterations_required,
                                                   double* mean_squared_iterations_required,
                                                   uint64_t* random_seeds) {
  int tid = threadIdx.x;
  int n_outer_loops = 1000;
  extern __shared__ int max_warp_iterations[];

  double mean_result = 0;
  double mean_square_result = 0;
  double accept_prob = 1.0-rejection_probabilities[blockIdx.x];

  for (int outer=0; outer<n_outer_loops; ++outer) {
    int num_iterations = 0;
    bool sample_not_obtained = true;
    while (sample_not_obtained) {
      double xi = prn(random_seeds+threadIdx.x+blockDim.x*blockIdx.x);

      // Surrogate rejection sampling:
      if (xi < accept_prob) sample_not_obtained = false;

      num_iterations++;
    }

    max_warp_iterations[tid] = num_iterations;

    // Now need to take the max num_iterations across the warp, because all threads wo
    // have waited on that last one. Parallel reduce using max as the binary operator.

    if (Warpsize > 64) { // this statement should evaluate at compile time
      __syncthreads();
      if (tid < 64) max_warp_iterations[tid] = max(max_warp_iterations[tid],
          max_warp_iterations[tid+64]);
    }
    if (Warpsize > 32) { // this statement should evaluate at compile time
      __syncthreads();
      if (tid < 32) max_warp_iterations[tid] = max(max_warp_iterations[tid],
          max_warp_iterations[tid+32]);
    }
    __syncthreads();
    if (tid < 16) max_warp_iterations[tid] = max(max_warp_iterations[tid],
        max_warp_iterations[tid+16]);
    __syncthreads();
    if (tid < 8) max_warp_iterations[tid] = max(max_warp_iterations[tid],
        max_warp_iterations[tid+8]);
    __syncthreads();
    if (tid < 4) max_warp_iterations[tid] = max(max_warp_iterations[tid],
        max_warp_iterations[tid+4]);
    __syncthreads();
    if (tid < 2) max_warp_iterations[tid] = max(max_warp_iterations[tid],
        max_warp_iterations[tid+2]);
    __syncthreads();
    if (tid == 0) {
```

```
      max_warp_iterations[tid] = max(max_warp_iterations[tid],
          max_warp_iterations[tid+1]);
      mean_result = (outer * mean_result + max_warp_iterations[0]) / (outer+1);
      mean_square_result = (outer * mean_square_result +
          max_warp_iterations[0]*max_warp_iterations[0]) / (outer+1);
    }
  }

  if (tid==0) {
    mean_iterations_required[blockIdx.x] = mean_result;
    mean_squared_iterations_required[blockIdx.x] = mean_square_result;
  }
}

int main(int argc, char* argv[]) {

  // Command line arguments are min rejection probability, max rejection probability
  if (argc != 4) {
    std::cout << "Incorrect argument count. First input min rejection probability, "
      "max, then the number of linearly spaced probabilities to check." << std::endl;
  }
  double min_r = std::stod(argv[1]);
  double max_r = std::stod(argv[2]);
  int n_probs = std::stoi(argv[3]);
  if (min_r < 0 or min_r > 1) std::cerr << "MIN_R INVALID" << std::endl;
  if (max_r < 0 or max_r > 1) std::cerr << "MAX_R INVALID" << std::endl;
  if (min_r >= max_r) std::cerr << "BADLY ORDERED PROBABILITIES" << std::endl;

  // Create the rejection probabilities to be used and move them to device
  double* dev_rej_probs;
  size_t nbytes = sizeof(double) * n_probs;
  std::vector<double> rejection_probabilities(n_probs);
  double dp = (max_r-min_r)/(n_probs-1);
  for (int i=0; i< n_probs; ++i)
    rejection_probabilities[i] = min_r + dp * i;
  cudaMalloc(&dev_rej_probs, nbytes);
  cudaMemcpy(dev_rej_probs, rejection_probabilities.data(),
      nbytes, cudaMemcpyHostToDevice);

  // Allocate memory for results on device
  double* mean_device;
  double* mean_sq_device;
  cudaMalloc(&mean_device, nbytes);
  cudaMalloc(&mean_sq_device, nbytes);

  // Allocate memory for random number seeds on device
  uint64_t* device_rngs;
  cudaMalloc(&device_rngs, sizeof(uint64_t) * n_probs * WARPSIZE);
  std::vector<uint64_t> rngs_host(n_probs * WARPSIZE);
  for (uint64_t i=0; i<n_probs*WARPSIZE; ++i)
    rngs_host[i] = master_seed + i;
  cudaMemcpy(device_rngs, rngs_host.data(),
      sizeof(uint64_t) * n_probs * WARPSIZE, cudaMemcpyHostToDevice);
```

```
  // Run the main kernel, one block per rejection probability
  calculate_mean_iterations_required<WARPSIZE><<<n_probs,
      WARPSIZE, sizeof(int)*WARPSIZE>>>(
      dev_rej_probs, mean_device, mean_sq_device, device_rngs);

  // Bring results back to host
  std::vector<double> mean_iterations(n_probs);
  std::vector<double> mean_squared_iterations(n_probs);
  cudaMemcpy(mean_iterations.data(), mean_device, nbytes, cudaMemcpyDeviceToHost);
  cudaMemcpy(mean_squared_iterations.data(), mean_sq_device, nbytes, cudaMemcpyDeviceT

  // And finally, print out the results
  for (int i=0; i<n_probs; ++i)
    std::cout << rejection_probabilities[i] << "␣"
              << mean_iterations[i] << "␣" << mean_squared_iterations[i] << std::endl;

  cudaFree(dev_rej_probs);
  cudaFree(mean_device);
  cudaFree(mean_sq_device);
  cudaFree(device_rngs);
}
```